Program #6
Due: Monday Apr 27th, 2015 at 11:30PM

Instructor	Dr. Stephen Perkins	
Office Location	ECSS 4.403	
Email Address	stephen.perkins@utdallas.edu	
Office Hours	Monday and Wednesday before class (7:30pm - 8:25pm) And by appointment	
TAs	Kenneth Joseph Platz	(kxp101120@utdallas.edu)
	Xuming Zhai	(xxz124830@utdallas.edu)
TA Office Location	Kenneth Platz Clark Center CN 1.202C	
	Xuming Zhai Clark Center CN 1.202D	
TA Office Hours	Kenneth Platz Tuesday/Thursday 12:00pm-2:00pm	
	Xuming Zhai Monday/Wednesday 5:00pm-6:00pm	

Purpose

Demonstrate the ability to download, build, install and use 3rd party library code distributed with GNU Automake (Curses Development Kit - CDK). Demonstrate the ability to create a multi-threaded application that implements a producer to multi-consumer model using the Boost C++ thread library. Demonstrate the ability to interact with non-thread safe libraries (CDK) by using mutexes. Demonstrate the ability to synchronize all threads by using barriers. Demonstrate the ability to control thread operation using semaphores. Demonstrate the ability to control groups of threads using the Boost **thread_group** class. Demonstrate the ability to use the C++ STL Vector class. Demonstrate the ability to visualize operation using a CDK matrix. Demonstrate the ability to control processes via UNIX signals.

Assignment

Overview

You will be creating a multi-threaded application along with a simple visualization for the running application. You are to create a program that displays a matrix on the screen. Each box in the matrix represents a bin. Initially, all the bins are empty. You are to spawn a separate consumer thread for each bin. The job of that consumer thread is to process any items that arrive in that bin. If no items are available for processing, the thread should block and wait for new items.

A separate thread, the producer thread, will be in charge of randomly placing items into the bins.

As consumer threads go through the process from startup, to consuming, to shutdown, they will update the on-screen matrix with their status (See ***Visualization*** below). This will give a live visualization of the running threads. In order to let the application run slow enough to watch it work, we will be relying on the sleep() function.

When the user is ready to stop the application, they will issue a Ctrl-C from the command line. This will instruct the producer to stop adding new items to bins and to start the process of shutting down the consumer threads in a controlled fashion. Each thread should finish all its work until its bin is empty. Then, it should exit. The main program should not exit until all threads have finished their work and exited (See ***Signals*** below)

Visualization

You will be using old-school UNIX tools to provide visualization of the working thread. Specifically, you will be using a third party terminal windowing library that is based on ncurses. Your graphical display will shows the states and operations of each consumer thread.

You will create a 5x5 display matrix using CDK. This display will represent a 5x5 table full of bins. As the producer thread places items into various bins, consumer threads will wake up to process those items. While processing, the consumer thread must update its specific CDK matrix box to display the following information:

1) The Thread identifier (index into parallel arrays is fine)
2) The state of the thread (***barrier, waiting, consuming, exiting***)
3) The quantity of items in the bin

CDK Library

The library you will use is called CDK. Download, build, and install the library from here:

http://invisible-island.net/datafiles/release/cdk.tar.gz

Like others we have used, this library is distributed in autoconf format. When you run the .configure script, I recommend you add the:

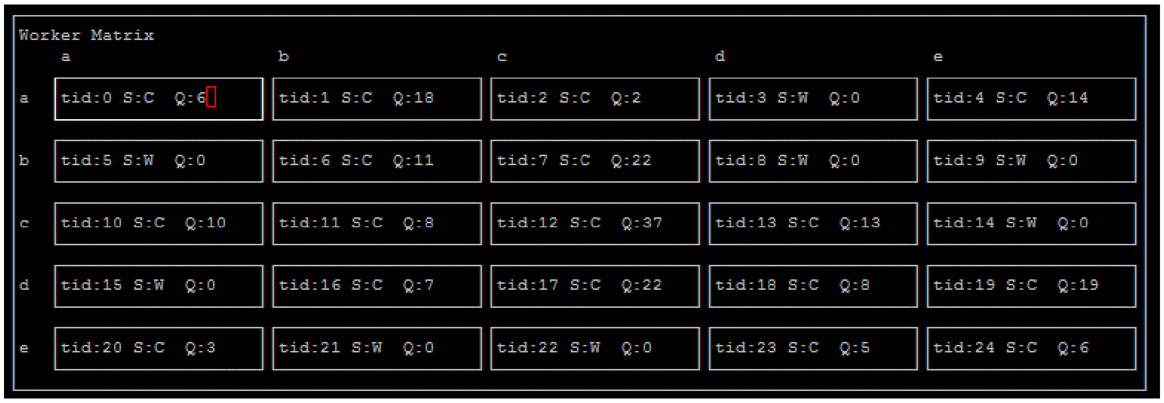
```
--prefix=<full path to your home folder>
```

When you ***make*** and ***make install***, the include files, man pages, and libraries will be installed in their respective subfolders under your home directory.

Once built, you will need to link against the Boost libraries, the CDK library, and the curses library using:

```
-lcurses -lcdk -lboost_thead-mt -lboost_system
```

Example Display



```
Worker Matrix
    a               b               c               d               e

a   tid:0 S:C  Q:6  tid:1 S:C  Q:18 tid:2 S:C  Q:2  tid:3 S:W  Q:0  tid:4 S:C  Q:14

b   tid:5 S:W  Q:0  tid:6 S:C  Q:11 tid:7 S:C  Q:22 tid:8 S:W  Q:0  tid:9 S:W  Q:0

c   tid:10 S:C Q:10 tid:11 S:C Q:8  tid:12 S:C Q:37 tid:13 S:C Q:13 tid:14 S:W Q:0

d   tid:15 S:W Q:0  tid:16 S:C Q:7  tid:17 S:C Q:22 tid:18 S:C Q:8  tid:19 S:C Q:19

e   tid:20 S:C Q:3  tid:21 S:W Q:0  tid:22 S:W Q:0  tid:23 S:C Q:5  tid:24 S:C Q:6
```

Each box shows the thread id (tid:), the thread's state (S:), and the quantity of items in the bin (Q:).

The Consumer Threads

Each bin has its own consumer thread. If a consumer thread's bin is empty, the thread will block while waiting on its semaphore. When items become available, semaphore will be signaled (post). The thread will wake up and take one item and "work on it". We will simulate the work by sleeping for 1 second. Once the work is done, the thread removes the item from the bin by decreasing the quantity of items by one. If more items remain (just wait on the semaphore), this process is repeated. If the bin is empty, the thread blocks until more items appear.

In the special case that a thread is signaled and it wakes to find its bin still empty (quantity is zero), it will take this as an indication that it should exit.

As consumer threads process their work, they are in charge of updating the quantity of work items in the bin and it is in charge of updating the value shown in the CDK matrix.

There will be 25 consumer threads (for a 5 x 5 matrix of bins).

Initialization barrier

Once started, threads should initialize themselves with any data they may need, and then they should wait on a barrier (State 'B' in ***Consumer Thread States*** below). Once all consumer threads (and the producer thread) have arrived at the barrier (signaling all are running and have finished initialization), then work may start.

Quantity

Each thread should maintain access to the quantity of items in its bin. This quantity is shared between the producer and consumer. The producer increases the quantity when it adds items and the consumer decreases the quantity when it removes items. Since it is a shared resource, it must be protected with a mutex.

Semaphore

Each thread should have its own semaphore. This synchronization primitive is how the producer and consumer threads communicate about when (and in some ways… how many) items are placed in the bin. When a producer adds ***n*** items to a bin, it increases the quantity count of the bin and then signals the worker thread's semaphore ***n*** times (posts n times). Since a semaphore is a counter, then means that the thread can process n times before blocking and waiting on the semaphore.

Mutex

Since the producer and the consumer must each update the quantity of items in the bin, you must be able to protect that value from joint updates. Each thread should have is own mutex that protects its quantity value.

Consumer Thread States

Consumer threads should update the visualization with the following states:

B	The thread is waiting on the initialization barrier
W	The thread is waiting for items to consume (waiting on its semaphore)
C	The thread is consuming an item
E	The thread has exited

Along with the state, the visualization should always show the quantity of items in each bin.

The Producer Thread

The producer can be its own thread or it can be the main program after it has spawned the consumer threads. For this document, I will refer to it as the producer thread regardless of whether it is spawned into a separate thread. There should only be one producer thread.

The producer thread's job is to randomly choose a bin, add a random amount of items to selected bin (between 10 and 20), and then sleep for 1 second. It then repeats this process infinitely. Adding work to a bin is done by increasing the quantity of items in the bin. The producer is in charge of updating the quantity of work items in the bin. It then must signal (post) the consumer thread's semaphore that new items have arrived. It should signal (post) once for each item added.

Like consumer threads, the producer should initialize itself and then wait on the barrier. In order to visualize that the threads are waiting at the barrier, the producer should sleep 5 seconds before arriving at its barrier. With a sleep of 5 seconds, it is likely that the producer will be the last to arrive at the barrier.

Once the producer has passed its barrier, it should again sleep for 5 seconds. This will allow us to see that all 25 threads have empty bins and are waiting on their semaphores. After the 5 seconds, the producer should start adding items to random bins once per second.

Thread Groups

All threads that are created should be added to a boost::thread_group. You can use the group methods to easily wait on all threads to exit (see **Signals**).

Signals

Your program should intercept and respond to the following UNIX signals:

SIGINT or SIGTERM - The daemon should remove the PID file, close logging, and exit.

When either of these signals is received (via a Ctrl-C on the keyboard), the producer should stop producing new items, the producer should signal threads that they should exit when the have finished all of their outstanding remaining work, the producer should then join on all threads and only exit when all consumer threads have finished.

Deliverables

You must submit your homework through ELearning. You must include a tarball that contains your Makefile, .h, .c, and any other important files. All source files and Makefiles need to have your name, email, and course number commented at the top. As usual, the code must compile and run on cs1.utdallas.edu.

Hints

- Create a set of parallel STL vectors for per bin quantity (int), per bin semaphore, and per bin mutex.
- When you start a thread, pass the thread an integer that represents its index into this set of parallel vectors.
- CDK is not thread safe. Create a separate mutex that you must lock before making screen updates.
- When threads have no work to perform, they wait on their semaphore. When the producer adds items, it wakes the threads and they consume the items. You may signal the semaphore (post) without increasing the quantity of items in a bin. If a thread awakens and the qty of items is zero, it can take this as an indication that it is time to exit.
- The CDK Matrix is based on an offset of 1 (not 0 like arrays). So, 1,1 is the first box.
- Your putty terminal must be large enough to show the matrix of you will get a segfault.
- You may use -Wno-write-strings to suppress const char * warnings.

Notes

Your code must be compiled using the **--Wall** compiler flag and must produce no errors/warnings.

No late homework is accepted.

Recommended Project Phases

This project is complex. In order to help with organization, students might consider using these phases as a guideline in how to write the code. Code should also be spread among separate files to keep code clean. Specifically:

Phase 1	• Create Makefile that compiles runs a hello world program o should use -I flags with include paths o should use -L with library paths o should use -l flags with libraries • verify backup target working (for backups and program submission)
Phase 2	• download/install CDK • Use CDK to draw a 5x5 matrix on the screen • Make sure you can update text in any box.
Phase 3	• Create a mutex to protect the screen updates. • Write a function that handles all updated to the CDK Matrix fields. • Protect the actual screen update by getting the lock on the mutex and then releasing when done • Only use this function to make updates • Possible Prototype: ***void update_cell(int xpos, int ypos, int thread_id, char thread_status, int qty);***
Phase 4	• Create a barrier variable that supports 26 items (25 consumers + 1 producer) • Spawn 25 threads and have them wait on the barrier • Put the thread ids into a thread_group • sleep for a few seconds • have the main program provide the final (26th) wait() on the barrier • When threads all exit, have main join on all the threads in the thread group before exiting.
Phase 5	• Have each spawned thread write its initial state to the matrix • Have each thread wait on the barrier • Have each thread wait on its own semaphore • Have the main program signal the barrier and the semaphore • When threads all exit, have main join on all the threads in the thread group before exiting.
Phase 6	• Fill in the code for the producer to add quantities to random bins • Fill in the code for the consumers to remove quantities from bins • If a consumer wakes and the quantity is zero, have the thread exit. • Flesh out the rest of the requirements

My implementation is all in a single file.